Taller de Arquitectura de Software

Refactorización de Monolito – Encuestas Espagueti

- Juan Sebastian Osorio Fierro
- Daniel Steve Fontalvo Matiz
- Leonado Fabio Perez Bermudez
- Juan Camilo Cruz Pardo

FASE 2 – Análisis de Anti-Patrones y Malas Prácticas

Objetivo

Analizar el sistema monolítico e identificar anti-patrones y malas prácticas clasificadas por:

- Seguridad
- Arquitectura
- SOLID
- Clean Code
- Manejo de errores
- Tipado
- Rendimiento

ACTIVIDAD 2.1 – Análisis del Backend (Spring Boot)

Archivo analizado:

EncuestaController.java

1. Seguridad

❌ 1.1 SQL Injection (Concatenación de SQL)

Ubicación

- Método `crear()`
- Método `encuesta()`
- Método `votar()`

Código Problemático

```
String sql = "INSERT INTO encuestas(pregunta, si_count, no_count) VALUES ('" +
pregunta + "', 0, 0)";
String sql = "SELECT id, pregunta, si_count, no_count FROM encuestas WHERE id = "
+ id;
```

🔎 Problema

Las consultas SQL se construyen mediante concatenación de strings.

Esto permite:

- Manipulación del query
- Alteración de la lógica SQL
- Inyección de código malicioso

⚠ Impacto

Vulnerabilidad crítica que compromete la seguridad del sistema.

❌ 1.2 Credenciales Hardcodeadas

```
private String URL = "jdbc:postgresql://db:5432/encuestas";
private String USER = "espagueti";
private String PASS = "espagueti";
```

- 🔎 Problema

1. Credenciales escritas directamente en el código fuente.
2. No se utilizan variables de entorno.
3. No se emplea application.properties o application.yml.

⚠ Impacto

- Riesgo de exposición de credenciales.
- Mala práctica DevOps.
- No es portable a producción.
- No cumple principios de configuración externa.

🏗 2. ARQUITECTURA

❌ 2.1 Ausencia de Arquitectura en Capas

El EncuestaController realiza:

1. Conexión a base de datos
2. Lógica de negocio
3. Validaciones
4. Construcción de SQL
5. Manejo de errores

- 🔎 Problema

No existen capas separadas como:

1. Controller
2. Service
3. Repository
4. DTO
5. Configuración de DataSource
6. Todo se encuentra en una sola clase.

⚠ Impacto

- Alta acoplación
- Difícil mantenimiento
- Baja escalabilidad
- Difícil testeo unitario
- Violación del principio de separación de responsabilidades

🧱 3. PRINCIPIOS SOLID

❌ 3.1 Violación del SRP (Single Responsibility Principle)

El controlador:

- Maneja lógica de negocio
- Gestiona conexión a BD
- Ejecuta SQL
- Valida datos
- Maneja errores
- Construye respuestas

🔎 Problema

Un controlador debería:

1. Recibir la request
2. Delegar la lógica al Service
3. Retornar la respuesta

- Actualmente hace múltiples responsabilidades.

⚠ Impacto

- Código difícil de mantener
- Baja cohesión
- Alto acoplamiento
- Difícil evolución futura

4. CLEAN CODE

❌ 4.1 Código Duplicado

Ejemplo de validaciones repetidas:

```
if (body == null || body.get("pregunta") == null ||
body.get("pregunta").toString().trim().length() < 3)
```

Este patrón se repite en múltiples métodos.

🎾 Problema

1. Validaciones repetidas.
2. No existe método reutilizable.
3. No se centraliza la lógica.

⚠ Impacto

- Mayor probabilidad de errores.
- Código más largo de lo necesario.
- Difícil mantenimiento. 6 🚨 5. MANEJO DE ERRORES ❌ 5.1 Uso de printStackTrace() y return null

```
catch (Exception e) {
    e.printStackTrace();
}
return null;
```

🎾 Problema

- Solo imprime el error en consola.
- Retorna null.
- No se devuelve un código HTTP adecuado.
- No se utiliza @ExceptionHandler.

⚠ Impacto

Respuestas inconsistentes. Mala práctica REST. Dificulta monitoreo y trazabilidad. Riesgo de NullPointerException en cliente.

🧮 6. TIPADO DEFICIENTE ❌ 6.1 Uso de Map sin Genéricos ni DTOs

```
public Map crear(@RequestBody Map body)
```

🎾 Problema

- Uso de Map sin tipado genérico.
- No existen DTOs.
- No hay validación estructurada.
- No hay clases de dominio.

⚠ Impacto

Pérdida de tipado fuerte. Mayor riesgo de errores en tiempo de ejecución. Código menos mantenible.

🚀 7. RENDIMIENTO ❌ 7.1 Conexión BD Recreada en Cada Llamada

```
private JdbcTemplate jdbc() {
    DriverManagerDataSource ds = new DriverManagerDataSource();
```

Cada vez que se ejecuta un endpoint:

1. Se crea un nuevo DataSource.
2. Se crea un nuevo JdbcTemplate.
3. Se genera una nueva conexión.

🎾 Problema

No se utiliza:

- Pool de conexiones
- Configuración centralizada de DataSource

⚠ Impacto

Ineficiencia Bajo rendimiento bajo carga No escalable

💻 ACTIVIDAD 2.2 – Análisis del Frontend (Angular)

Archivos analizados:

- crear.component.ts
- encuesta.component.ts
- respuestas.component.ts

❌ 1. URL Hardcodeada

this.http.post("http://localhost:8081/crear", ...)

🔎 Problema

No se usa environment.ts. No hay configuración por entorno.

⚠ Impacto

No portable a producción. Mala práctica de configuración.

❌ 2. Uso Directo de HttpClient en el Component constructor(private http: HttpClient)

🔎 Problema

No existe una capa Service intermedia. Se viola el patrón de arquitectura Angular recomendado.

⚠ Impacto

Violación del patrón Service. Lógica mezclada con presentación. Dificulta pruebas unitarias.

❌ 3. Manipulación Directa del DOM document.getElementById(...)

🔎 Problema

Angular debe utilizar:

- Data Binding
- Directivas
- Templates reactivos
- Manipular el DOM directamente es un anti-patrón.

⚠ Impacto

Código menos mantenible. Rompe el enfoque declarativo de Angular.

❌ 4. Uso Excesivo de any (r: any)

🔎 Problema

No hay interfaces tipadas. No hay modelos de datos. Se pierde el beneficio de TypeScript.

⚠ Impacto

Riesgo de errores. Pérdida de autocompletado fuerte. Código menos robusto.

❌ 5. Polling Manual con setInterval

Uso de:

- setInterval(...)

En lugar de:

- Observables
- RxJS
- AsyncPipe

⚠ Impacto

Código menos reactivo. No aprovecha el paradigma Angular. Posible consumo innecesario de recursos.

🟫 8. Instrumentos de Casos de Uso – Impacto Arquitectónico

8.1 Actores Identificados

Actor	Descripción
Usuario (Votante)	Persona que crea encuestas y emite votos
Sistema Backend	API REST que procesa las solicitudes
Base de Datos	PostgreSQL que almacena encuestas y votos

8.2 Casos de Uso Principales

ID	Caso de Uso	Endpoint	Método
CU-01	Crear Encuesta	`/crear`	POST
CU-02	Listar Encuestas	`/encuestas`	GET
CU-03	Consultar Encuesta	`/encuesta/{id}`	GET
CU-04	Votar Encuesta	`/votar`	POST

🔎 8.3 Análisis de Impacto por Caso de Uso

CU-01 – Crear Encuesta

Endpoint: `POST /crear`
Archivo: `EncuestaController.java`
Método: `crear()`

Flujo Normal

1. Usuario envía pregunta.
2. Sistema valida longitud.
3. Inserta encuesta en base de datos.
4. Retorna ID y pregunta creada.

Anti-Patrones Detectados

- SQL concatenado (vulnerabilidad SQL Injection).
- Uso de `Map` sin tipado.
- Validaciones repetidas manualmente.
- Uso de `return null` ante error.
- Creación manual de `DataSource`.

Impacto Arquitectónico

- Riesgo crítico de seguridad (inyección SQL).
- Alta acoplación a infraestructura.
- Baja mantenibilidad.
- Difícil escalabilidad futura.

CU-02 – Listar Encuestas

Endpoint: `GET /encuestas`
Método: `encuestas()`

Flujo Normal

1. Sistema consulta todas las encuestas.
2. Retorna lista ordenada por ID descendente.

Anti-Patrones Detectados

- Validación innecesaria de URL dentro del método.
- Retorno `null` ante error.
- Re-creación de conexión en cada llamada.

Impacto Arquitectónico

- Ineficiencia por recreación de conexiones.
- Riesgo de respuestas inconsistentes.
- Violación del principio SRP.

▧ CU-03 – Consultar Encuesta

Endpoint: `GET /encuesta/{id}`
Método: `encuesta()`

Flujo Normal

1. Usuario envía ID.
2. Sistema consulta encuesta específica.
3. Retorna datos con conteos.

Anti-Patrones Detectados

- SQL concatenado con ID.
- Validación manual repetida.
- Manejo deficiente de errores.

Impacto Arquitectónico

- ▩ Vulnerabilidad potencial.
- ▨ Código duplicado.
- ▨ Falta de contrato de error HTTP.

▧ CU-04 – Votar Encuesta

Endpoint: `POST /votar`
Método: `votar()`

Flujo Normal

1. Usuario envía ID y voto (SI/NO).
2. Sistema actualiza contador.
3. Retorna encuesta actualizada.

Anti-Patrones Detectados

- SQL concatenado en UPDATE.
- Uso de `Map` sin DTO.
- Lógica condicional rígida.
- Repetición de consulta final.

Impacto Arquitectónico

- ▩ Riesgo de manipulación de datos.
- ▨ Alta duplicación de lógica.
- ▨ Falta de extensibilidad (si se agregan más tipos de voto).

🔗 8.4 Relación Sistémica de Anti-Patrones

Los problemas identificados no son aislados.

La ausencia de arquitectura en capas provoca:

- SQL dentro del Controller.
- Creación manual de conexiones.
- Validaciones repetidas.
- Manejo de errores inconsistente.

El uso de `Map` sin tipado genera:

- Validaciones manuales repetitivas.
- Falta de contratos explícitos.
- Mayor probabilidad de errores en tiempo de ejecución.

Esto demuestra que el sistema presenta un problema estructural de diseño arquitectónico, no simplemente errores puntuales de implementación.

📊 8.5 Matriz de Impacto por Caso de Uso

Caso de Uso	Seguridad	Arquitectura	Mantenibilidad	Escalabilidad
CU-01 Crear	🔴 Alta	🔴 Alta	🔴 Alta	🟡 Media
CU-02 Listar	🟡 Media	🔴 Alta	🔴 Alta	🟡 Media
CU-03 Consultar	🔴 Alta	🔴 Alta	🔴 Alta	🟡 Media
CU-04 Votar	🔴 Alta	🔴 Alta	🔴 Alta	🔴 Alta

📌 Conclusión de Instrumentos de Casos de Uso

Cada caso de uso del sistema se encuentra afectado por anti-patrones estructurales que comprometen:

- Seguridad (SQL Injection).
- Separación de responsabilidades.
- Rendimiento bajo carga.
- Evolución futura del sistema.

CONCLUSIÓN TÉCNICA – FASE 2

- El sistema presenta múltiples anti-patrones y malas prácticas:
- Vulnerabilidad a SQL Injection
- Credenciales hardcodeadas

- Violación del principio SRP
- Ausencia de arquitectura en capas
- Manejo deficiente de errores
- Bajo rendimiento por mala gestión de conexiones
- Uso incorrecto de patrones en Angular
- Se trata de un monolito funcional pero mal estructurado, lo que justifica una refactorización arquitectónica hacia:

1. Arquitectura en capas (Controller → Service → Repository)
2. Uso de DTOs
3. Configuración externa
4. Uso de Prepared Statements
5. Implementación de pool de conexiones
6. Separación adecuada en frontend (Service layer)
